Building high performance, fully concurrent garbage collectors with confidence

Richard Jones
cs.kent.ac.uk/~rej

University of Kent | Computing

Outline

- Overview of GC algorithms
- Case study: Sapphire on-the-fly replication collector
- Abstractions and invariants are essential for comprehension
- Design pattern for phase transitions
- Copying with transactional memory
- Model checking GC components

Collaborators

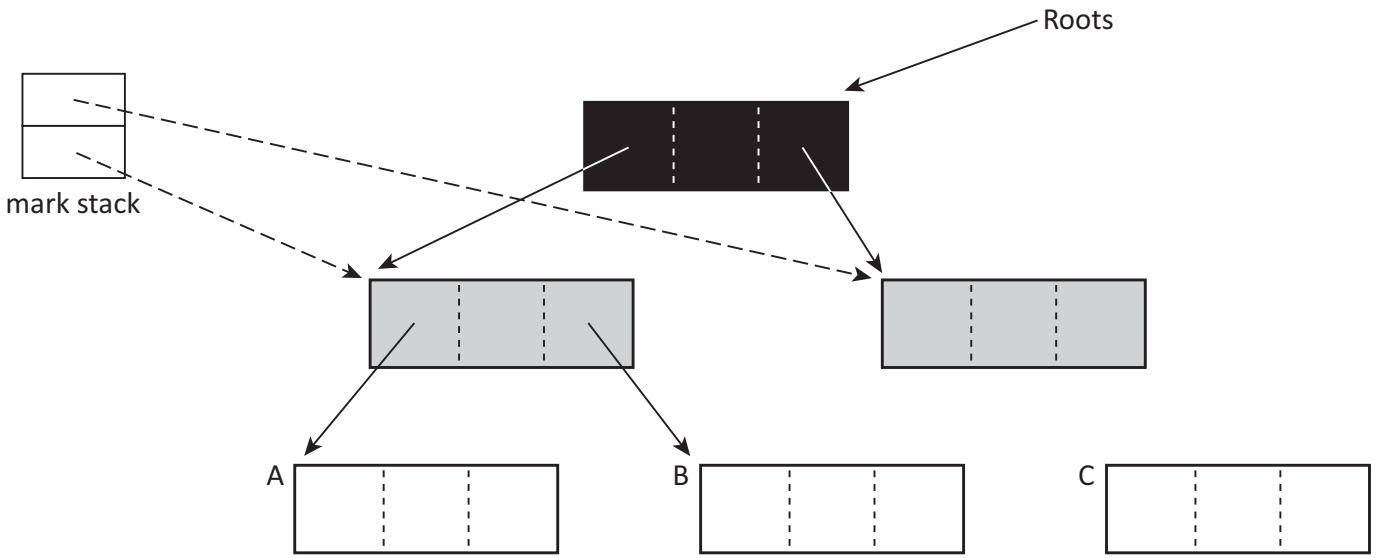
Carl Ritson
cs.kent.ac.uk/~cgr

Tomoharu Ugawa
spa.info.kochi-tech.ac.jp/~ugawa/index-e.html

GC BACKGROUND

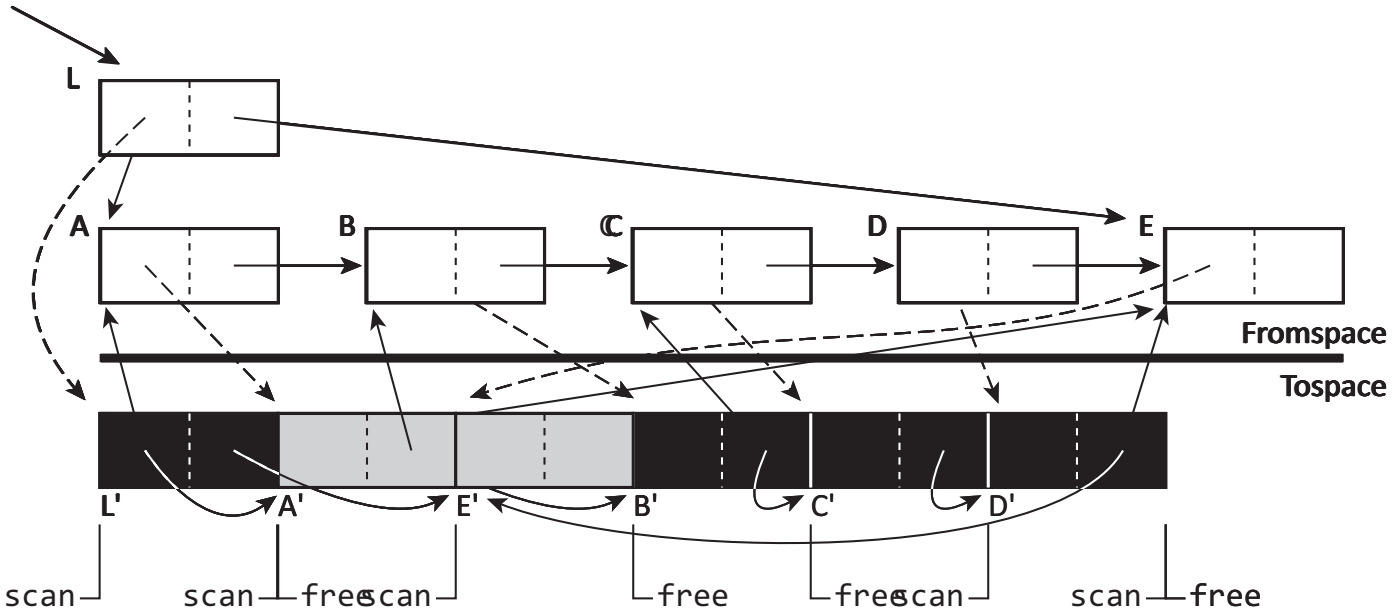
Styles of tracing collection

- Mark-sweep
- Copying GC

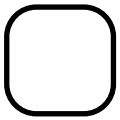


Styles of tracing collection

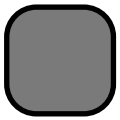
- Mark-sweep
- Copying GC [Cheney, CACM, 1970]



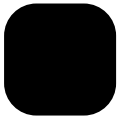
Tricolour abstraction



not yet been reached by the GC.



visited by the GC,
but fields need to be scanned.

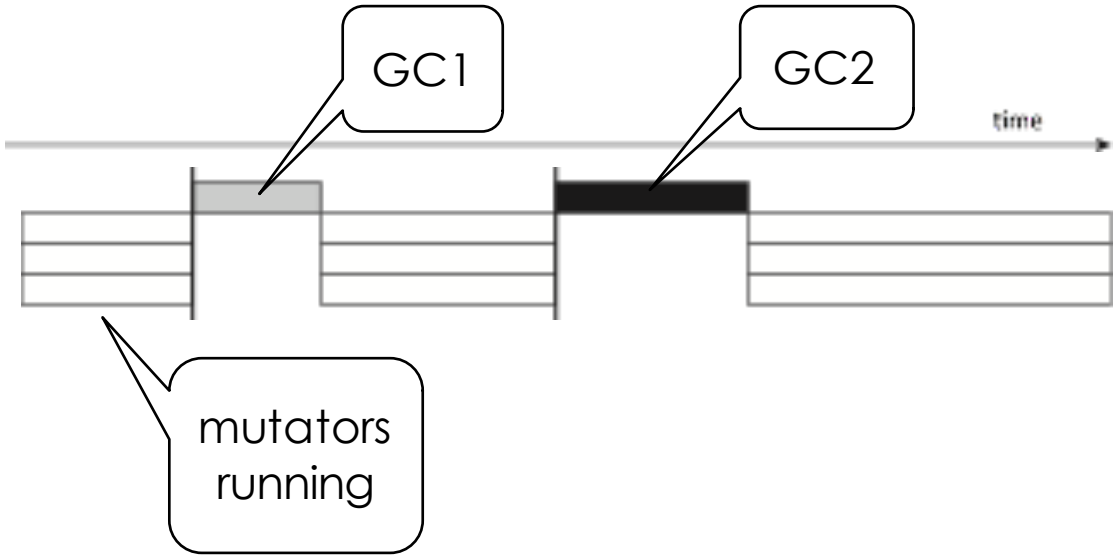


visited by the GC,
all fields have been scanned.

PARALLEL, INCREMENTAL and CONCURRENT GC

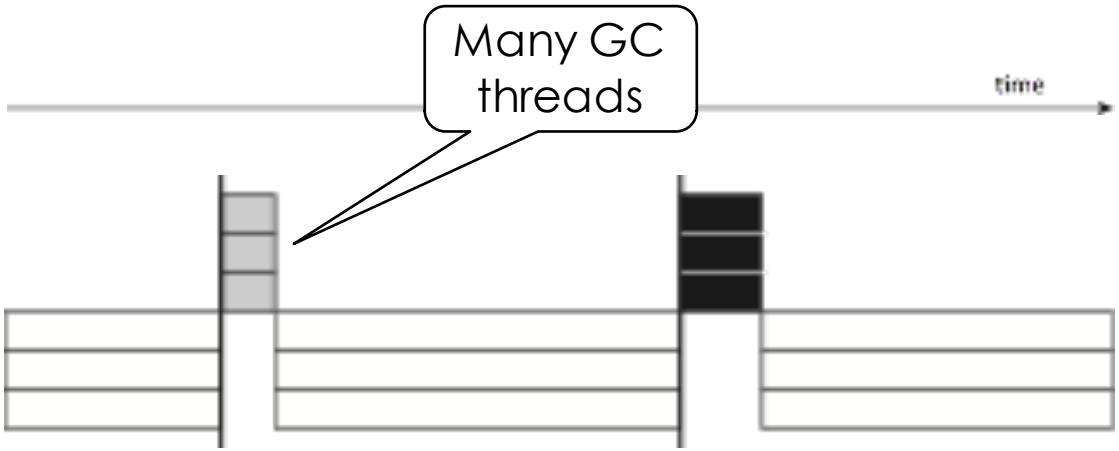
Parallelism

- Stop the world (STW)
- Parallel GC
- Incremental GC
- Mostly concurrent GC
- On-the-fly (OTF) GC



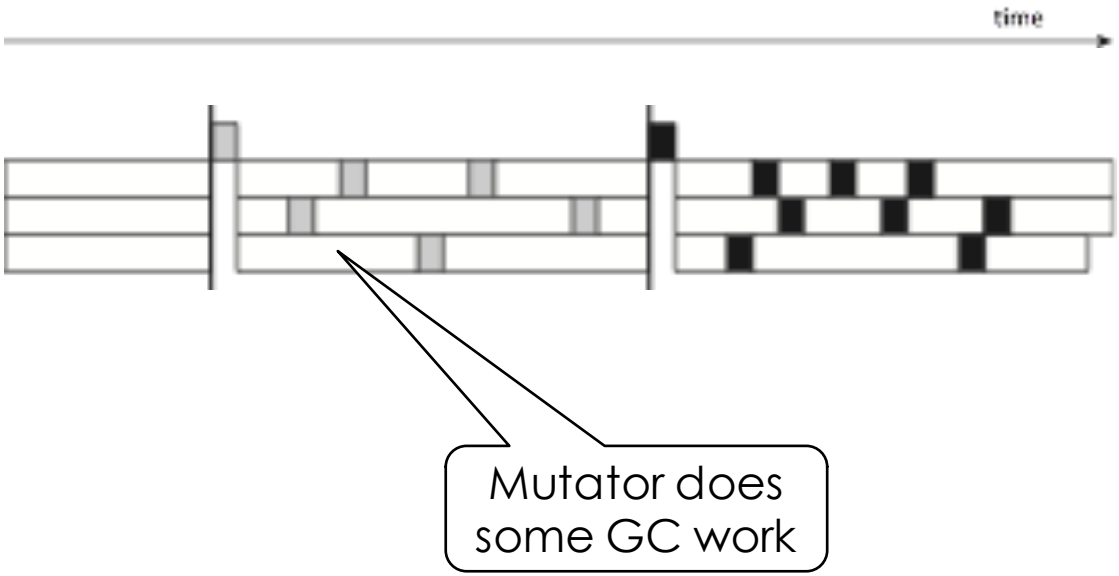
Parallelism

- Stop the world (STW)
- Parallel GC
- Incremental GC
- Mostly concurrent GC
- On-the-fly (OTF) GC



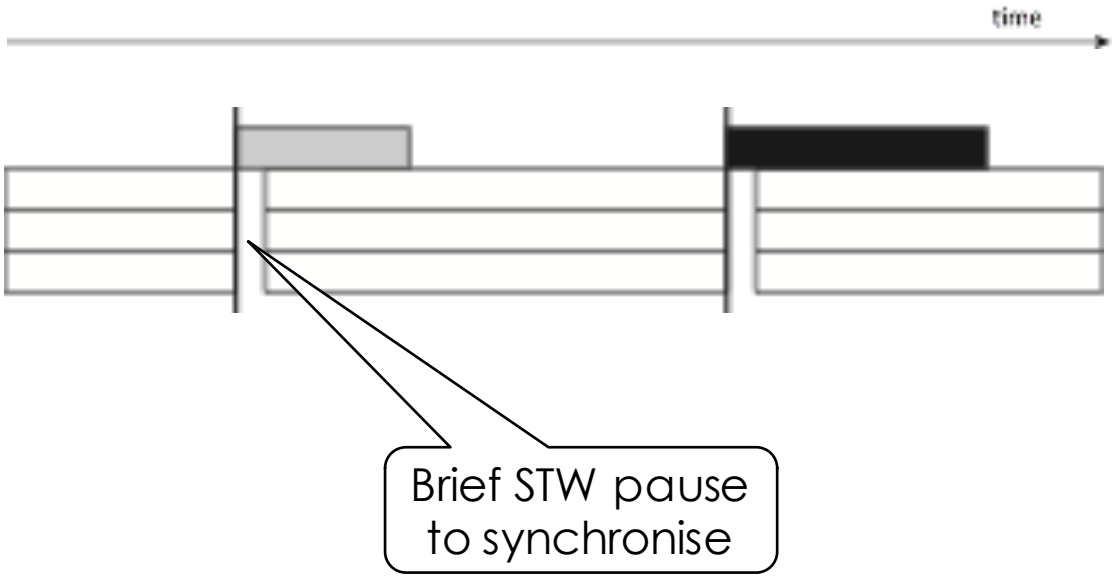
Parallelism

- Stop the world (STW)
- Parallel GC
- Incremental GC
- Mostly concurrent GC
- On-the-fly (OTF) GC



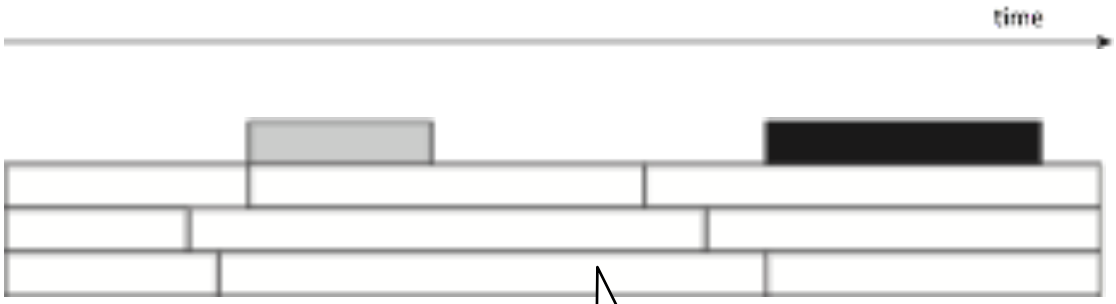
Parallelism

- Stop the world (STW)
- Parallel GC
- Incremental GC
- Mostly concurrent GC
- On-the-fly (OTF) GC



Parallelism

- Stop the world (STW)
- Parallel GC
- Incremental GC
- Mostly concurrent GC
- On-the-fly (OTF) GC



No STW pauses but 1 thread at at time halted

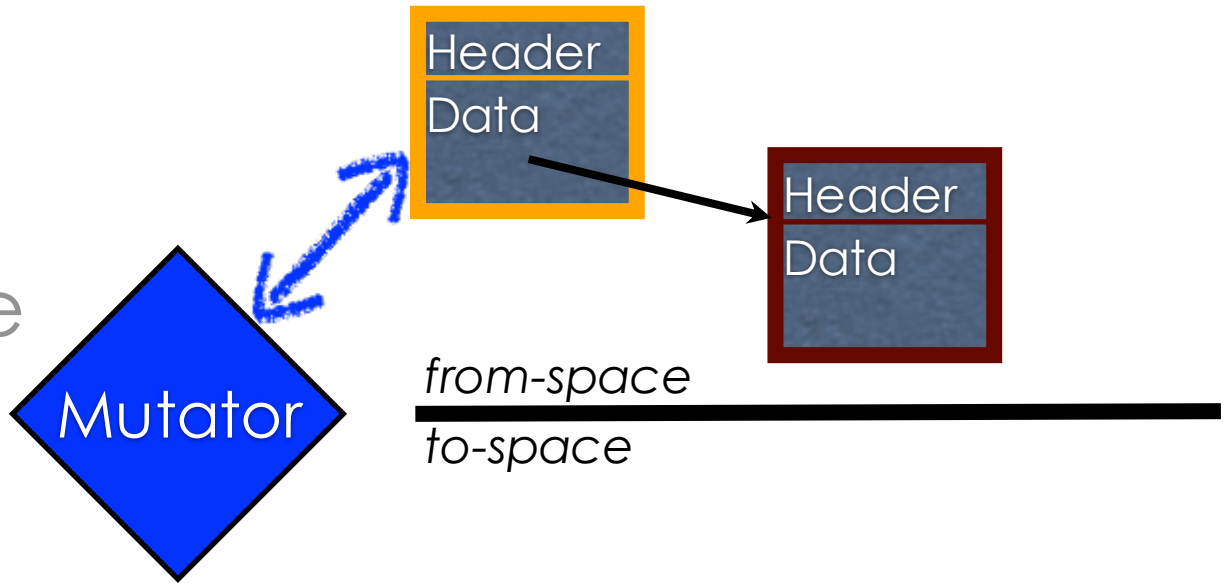
SAPPHIRE

Sapphire

- On-The-Fly concurrent replicating GC
  [Hudson & Moss, Concurrency Practice & Experience, 2003]
  - OTF: never stop more than one thread at a time
  - Copying: fast allocation, eliminate fragmentation
  - Synchronisation: GC pays (mostly)
- New implementation in Jikes RVM [jikesrvm.org]
  - Research JVM, "easy" to replace components
  - Largely written in Java

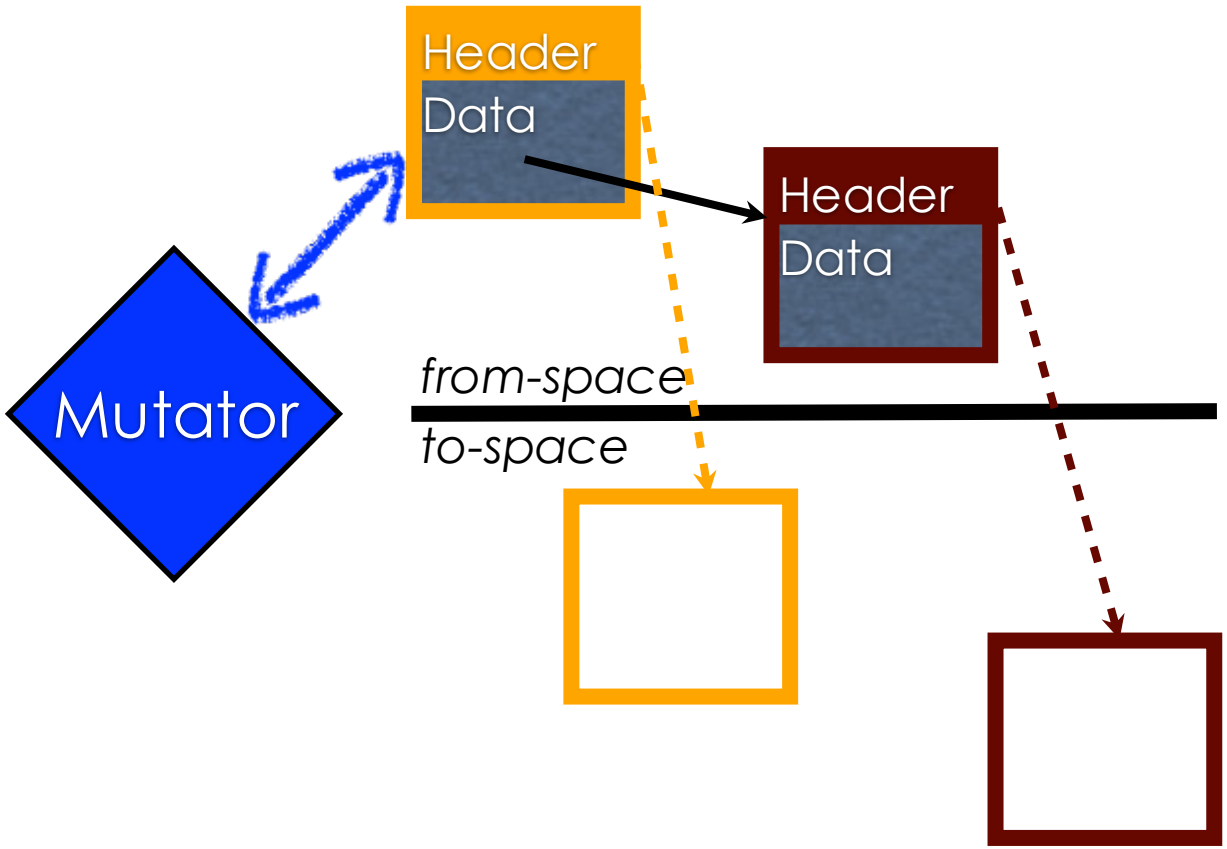
Phases

- Before
- Mark phase
- Copy phase
- Flip phase
- After



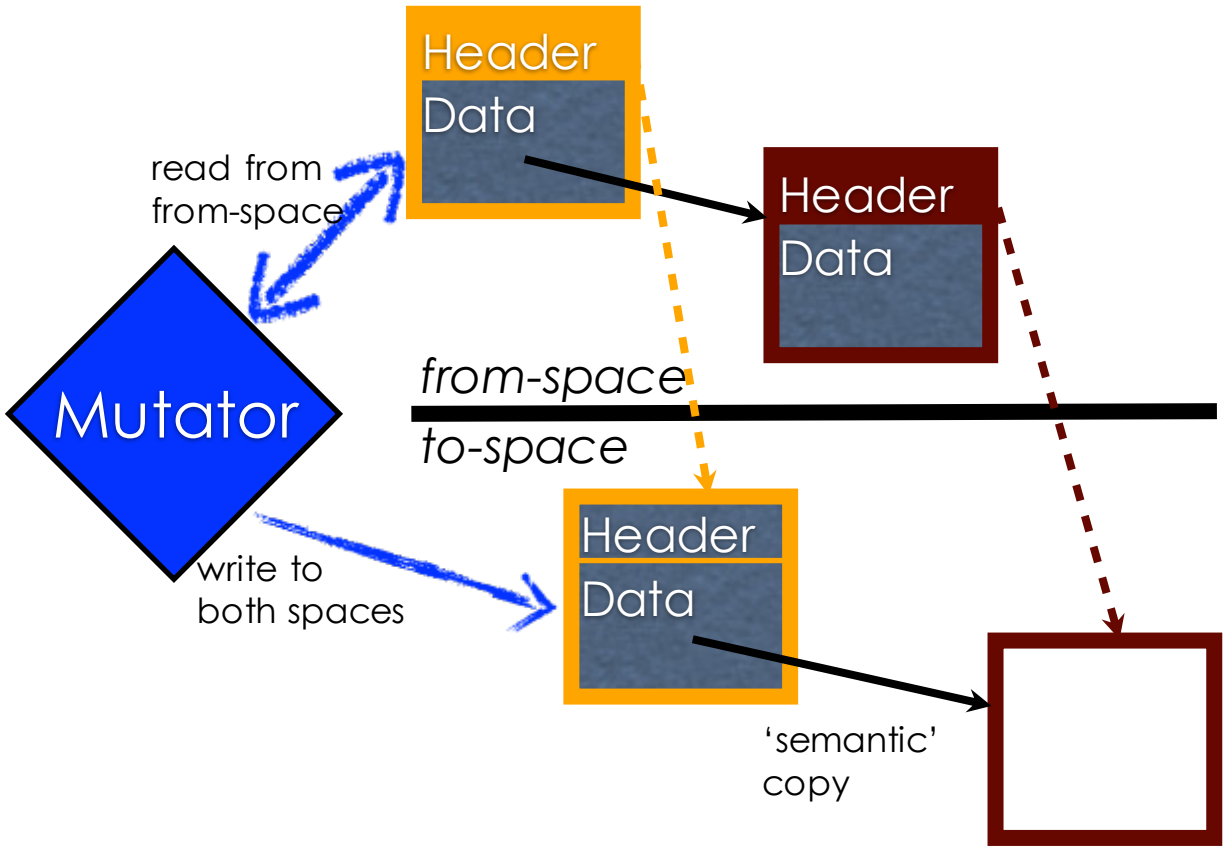
Phases

- Before
- Mark phase
- Copy phase
- Flip phase
- After



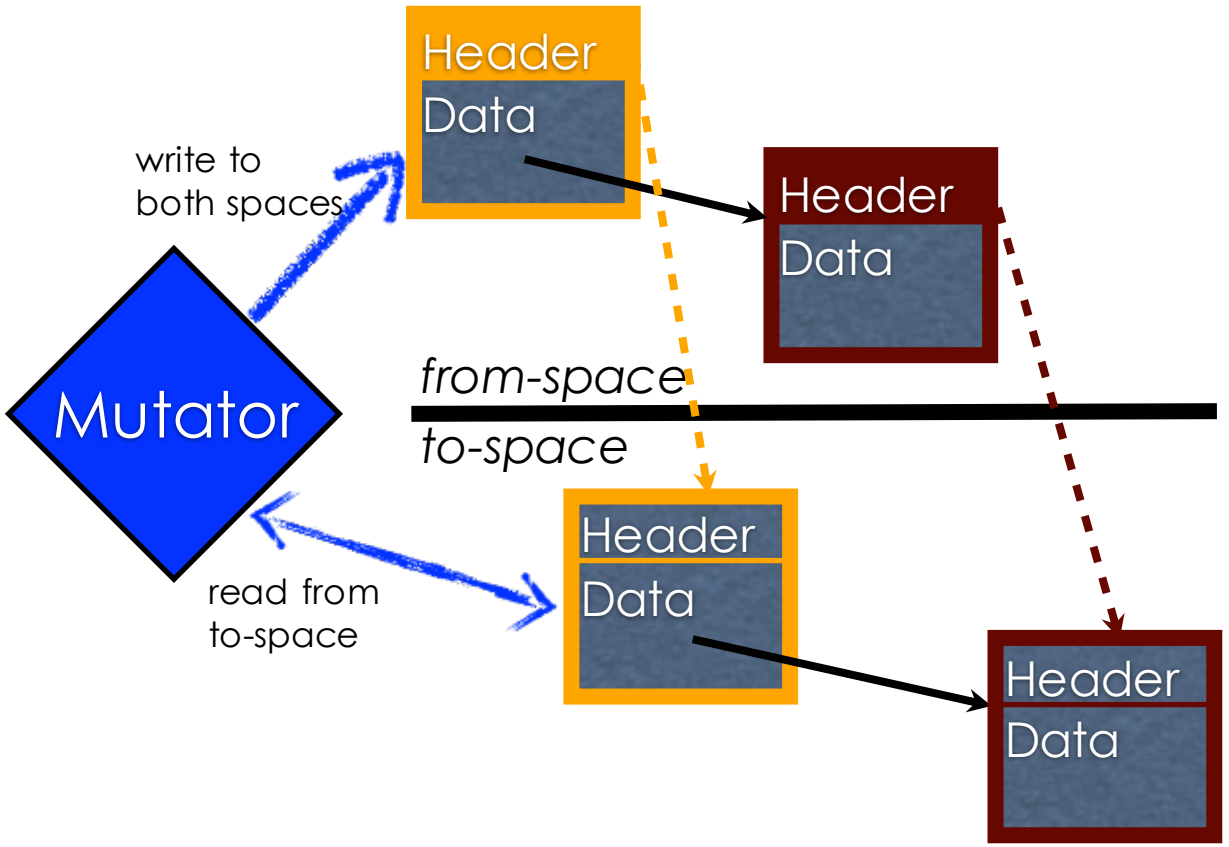
Phases

- Before
- Mark phase
- Copy phase
- Flip phase
- After



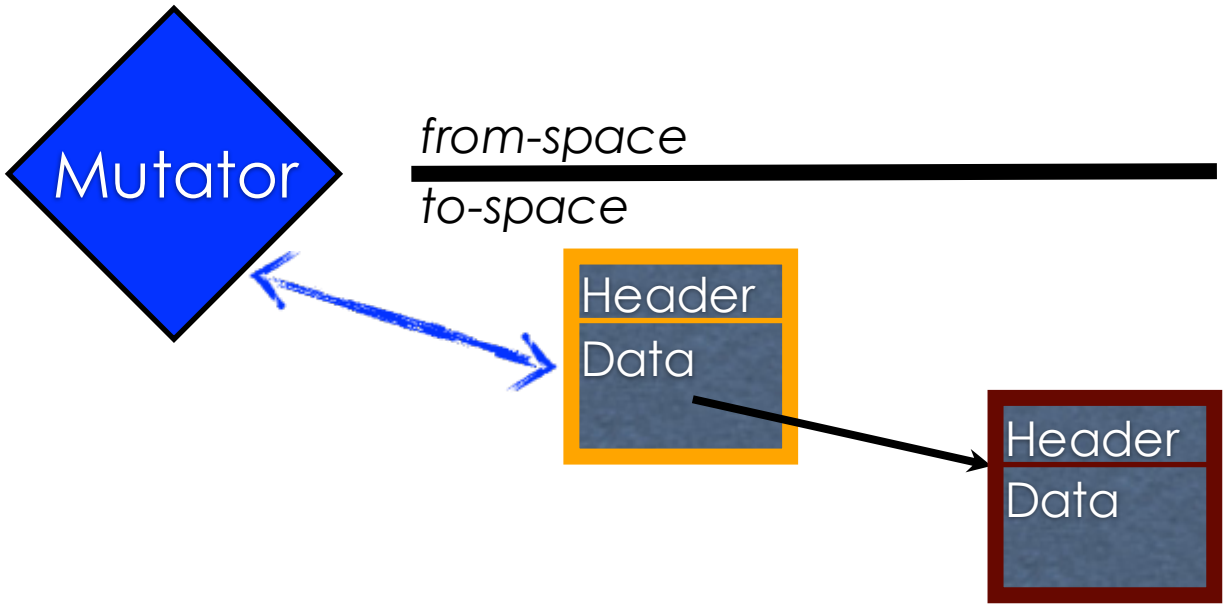
Phases

- Before
- Mark phase
- Copy phase
- Flip phase
- After



Phases

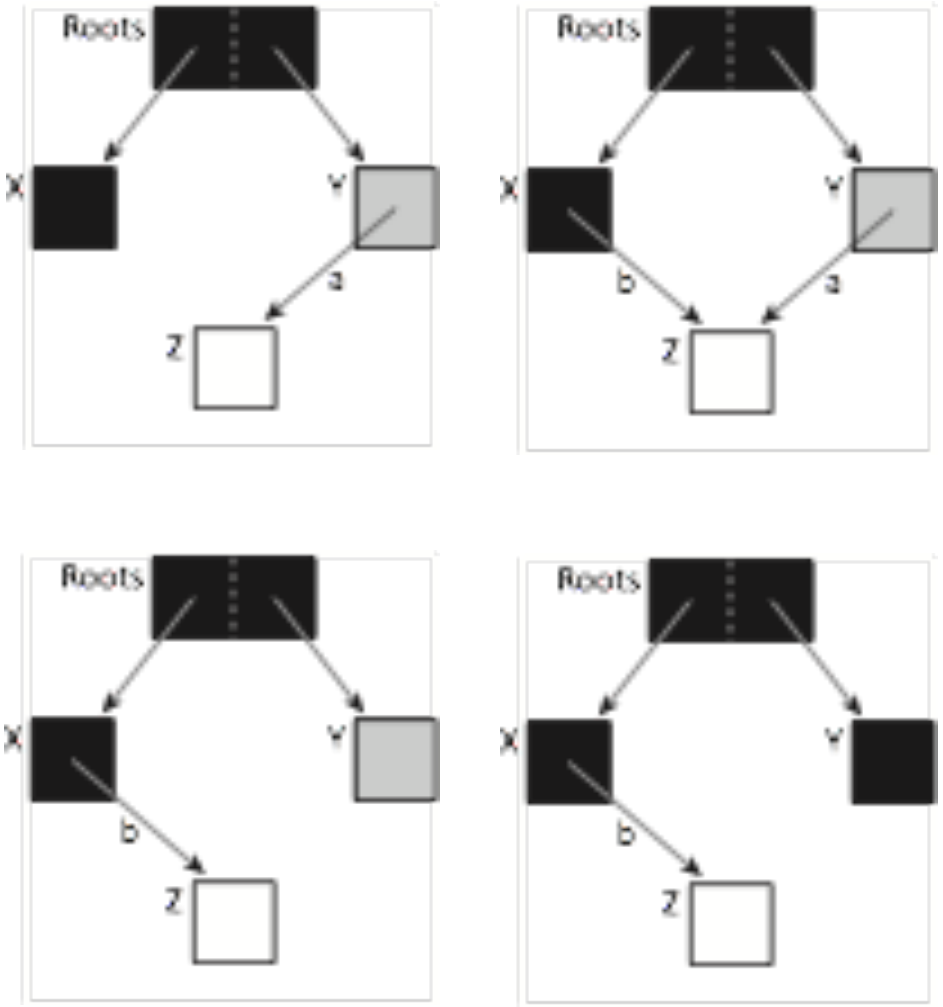
- Before
- Mark phase
- Copy phase
- Flip phase
- After



WHAT COULD POSSIBLY GO WRONG?

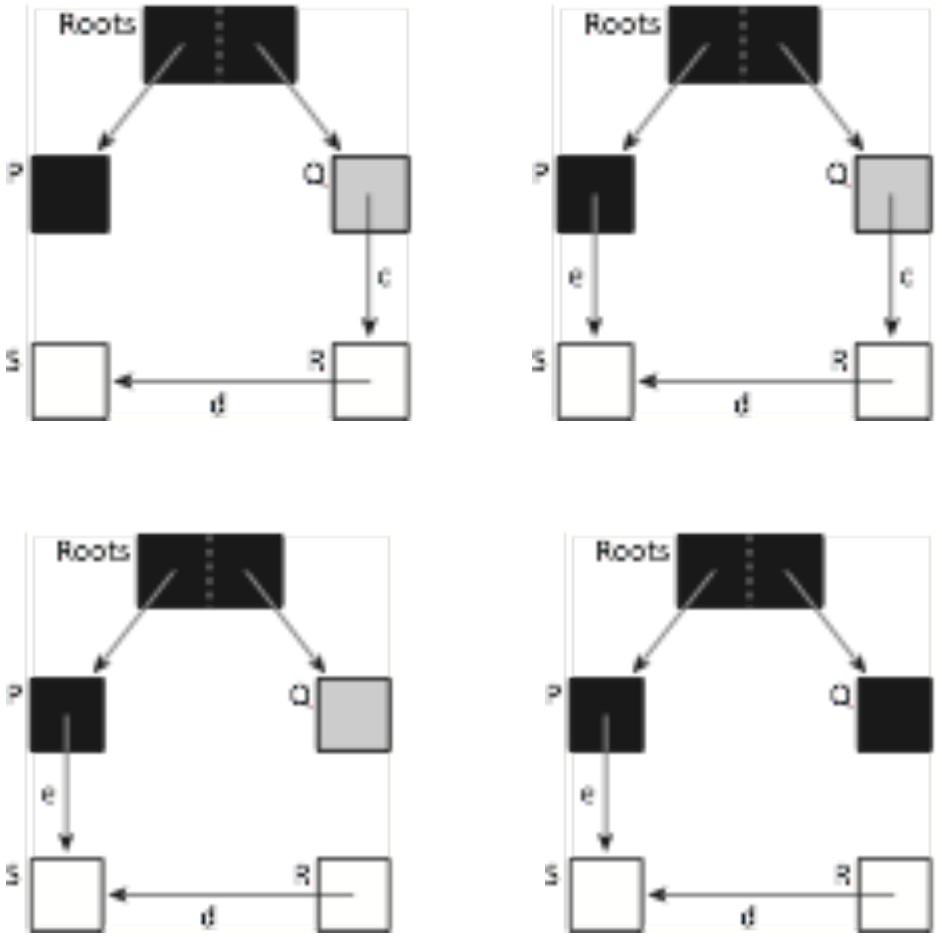
The lost object problem

- Mutator 'hides' a reachable object from GC



The lost object problem

- Mutator 'hides' a *transitively* reachable object from GC



Tricolour abstraction to the rescue!

Wilson conditions [Wilson, IWMM92]

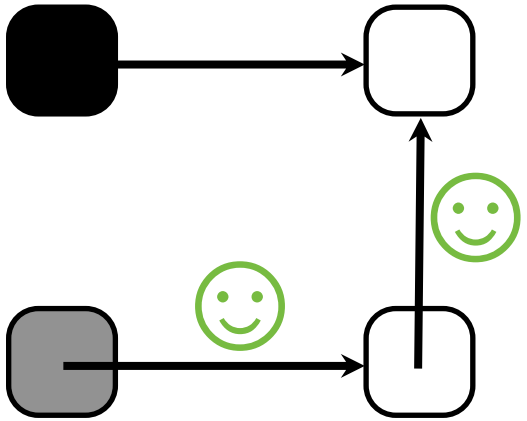
- Unsafe collection of a reachable object *if and only* if
  1. A mutator stores a pointer to a white object into a black object
- *AND*
  2. All paths from any grey objects to that white object are destroyed.
- Enforce invariants to prevent one or other of these conditions from arising....

Tricolour invariants

- **Strong invariant**:
  No pointers from black to white objects.

Prevent Cond. 1

- **Weak invariant**:
  All white objects pointed to by a black object are reachable from some grey object, either directly or through a chain of white objects ("grey protected").



Prevent Cond. 2

Barriers to maintain invariants

- Maintain invariants with *barriers* on writes (or reads).
  - A mutator action that communicates with the collector. Typically, some code added around a pointer read or write (e.g. putfield bytecode).

- Insertion (incremental update) write barrier.
- Deletion (snapshot at the beginning) write barrier.

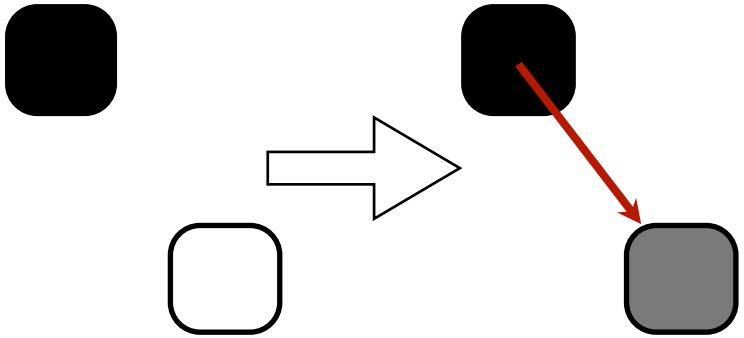
Mutator colours [Pirinen, ISMM98]

- Grey mutator:
  roots still to be traced, or need to be rescanned
  - ➡ may point to black, grey or white objects

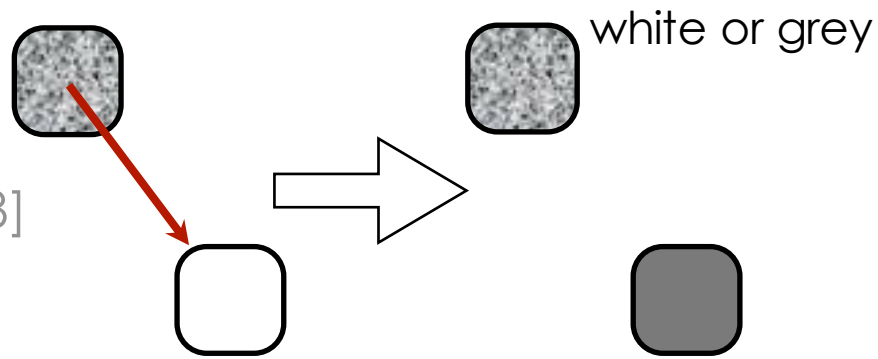
- Black mutator:
  roots have been traced, will not be rescanned
  - ➡ (strong invariant) will *never* point to white objects
  - ➡ (weak invariant) may point to *grey protected* white objects

Mutator techniques

- Grey mutator:
  - strong invariant — insertion write barrier [Steele, CACM75; Dijkstra, 1976, CACM78]



- Black mutator:
  - weak invariant — deletion write barrier [Abraham & Patel, ICPP87; Yuasa, JSS90]
  - strong invariant — read barrier [Baker, CACM78]



TERMINATION and INITIALISATION

Termination of trace

Grey mutator

- Trace until no grey objects remain.
  // Mutators may hold grey or white references.
- Loop:
  Scan each thread,
  shading any white object found.
  If any grey objects exposed
  then trace grey objects.
  Else break.

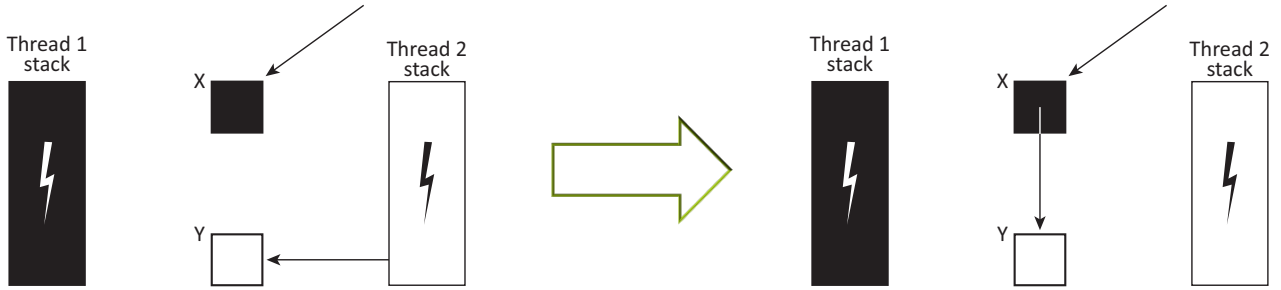
Black mutator

- Trace until no grey objects remain.

Even with the weak tricolour invariant the mutator can contain only black references.
- There are no grey objects.
- So there are no white objects reachable from grey objects.
- Because the mutator is black there is no need to rescan its roots.

Initialisation of trace

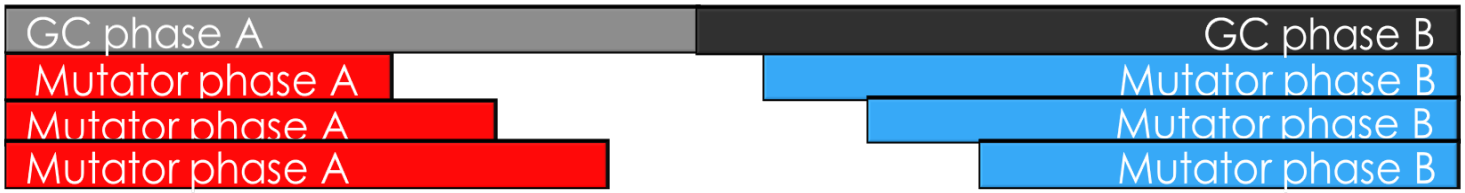
- Mostly concurrent collectors…
  - Stop the world briefly to scan mutator thread stacks
  - Can use insertion or deletion write barrier, e.g.
    black mutator / deletion barrier / allocate new objects black
- On-the-fly collectors
  - Deletion barrier not sufficient since…
  - Mixture of black (scanned) and white (unscanned) stacks
  - Must use insertion and deletion barrier until all roots scanned



PHASE TRANSITIONS

Stop-the-world / Mostly concurrent collection

- Phase changes are synchronous
- Once the collector has entered a new phase B, no mutator observes a GC state S_A corresponding to the previous phase A.
- At any time, all mutators have a coherent GC state — all observe S_A or all observe S_B.



On-the-fly collection

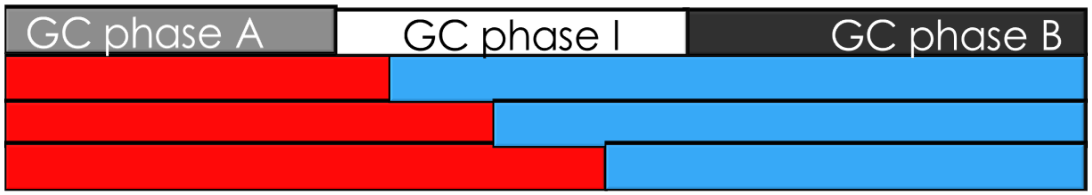
- Phase changes are ragged.
- A mutator will not recognise that the phase has been changed until it reaches a GC-safe point.
- Consequences:
  1. Collectors cannot start work until it has determined that all mutators have recognised that the phase has been changed.
  2. Different mutators may observe different GC states. Invariants required by different GC states may be incompatible!

[Doligez and Gonthier, POPL94]

Phase Transition Design Pattern

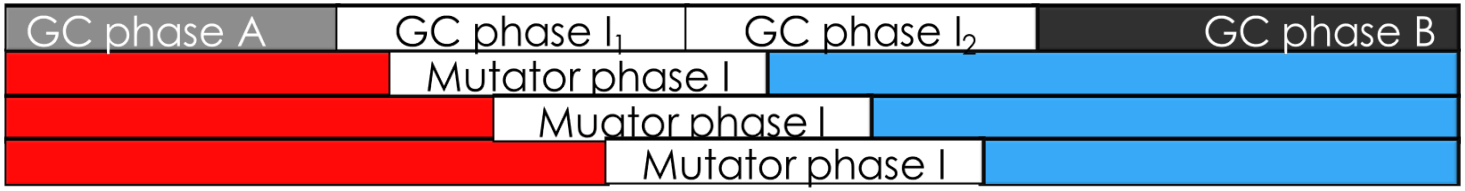
Type I: "recognising phase change"

Insert an intermediate GC phase I:
GC simply handshakes with every mutator at their GC-safe points



Type II: "incompatible invariants"

Insert two GC intermediate phases I_1 and I_2;
write barrier for an intermediate phase is typically more complex



TRANSACTIONAL SAPPHIRE

Intel's Haswell

- Transactional Memory Extensions (TSX-NI)
  - (Restricted) Transactional Memory
  - ... with limited processor support

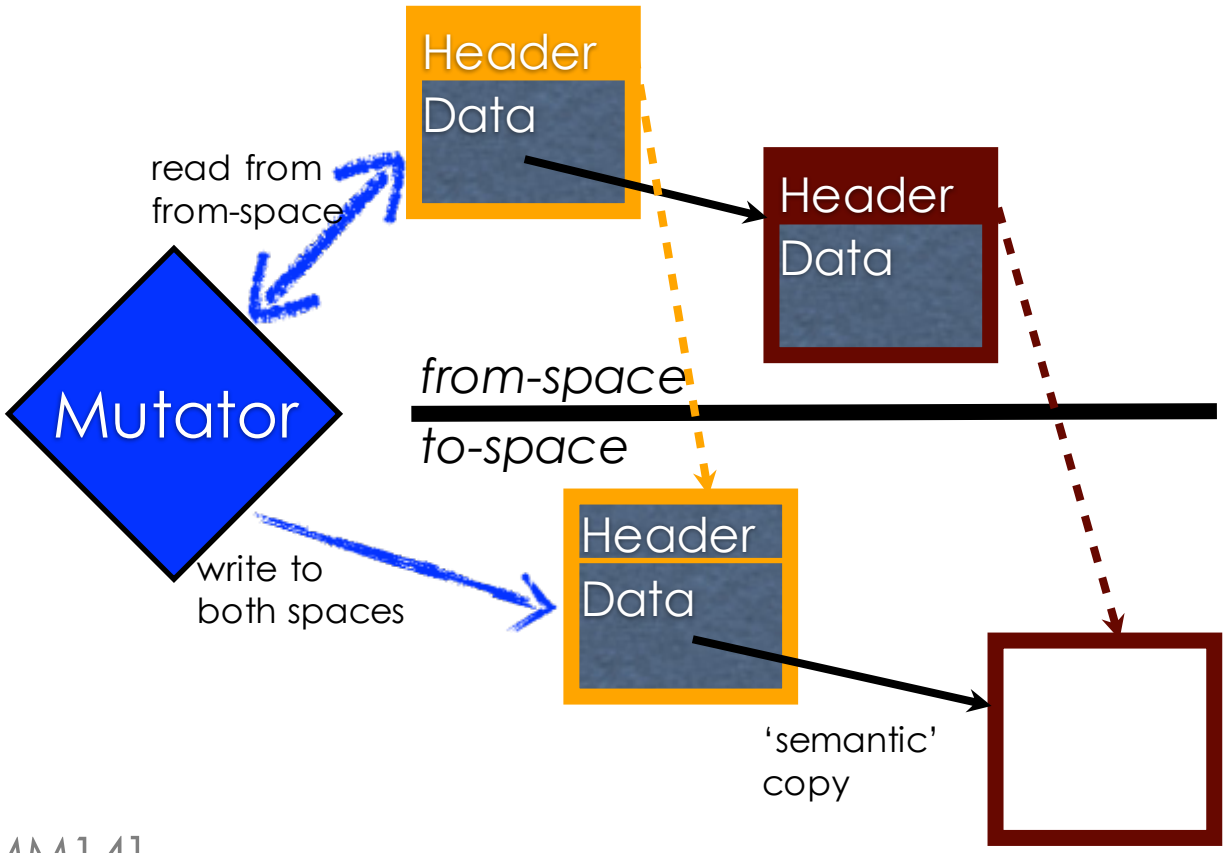
- XBEGIN ... XEND
  - Up to ~16KiB of read and writes

Complexities

- Setup of transaction is expensive (3x CAS)
- Fallback required if transaction fails
- Aborted transactions are expensive

Sapphire copying

Mutator must write to both replicas



[Ritson et al, ISMM14]

CAS

```
u = load (to-space)
v = load (from-space)
if (u != v)
        compare-and-swap  (to-space, u, v)
        restart
else
        break
```

Unsafe

v = load (from-space)

store (to-space, v)

HTM

XBEGIN

v_1 = load (from-space)

store (to-space, v_1)

v_2 = load (from-space)

store (to-space, v_2)

...

XEND

If we fail...
CAS

MHTM

XBEGIN

copy object 1

copy object 2

...

copy object n

XEND

Copy several objects in a single transaction

Planned MHTM

scan and record object 1 .. n

XBEGIN

copy object 1

copy object 2

...

copy object n

XEND

Take as much work as possible out of the transaction

STM

v_1 = load (from-space)

store (to-space, v_1)

v_2 = load (from-space)

store (to-space, v_2)

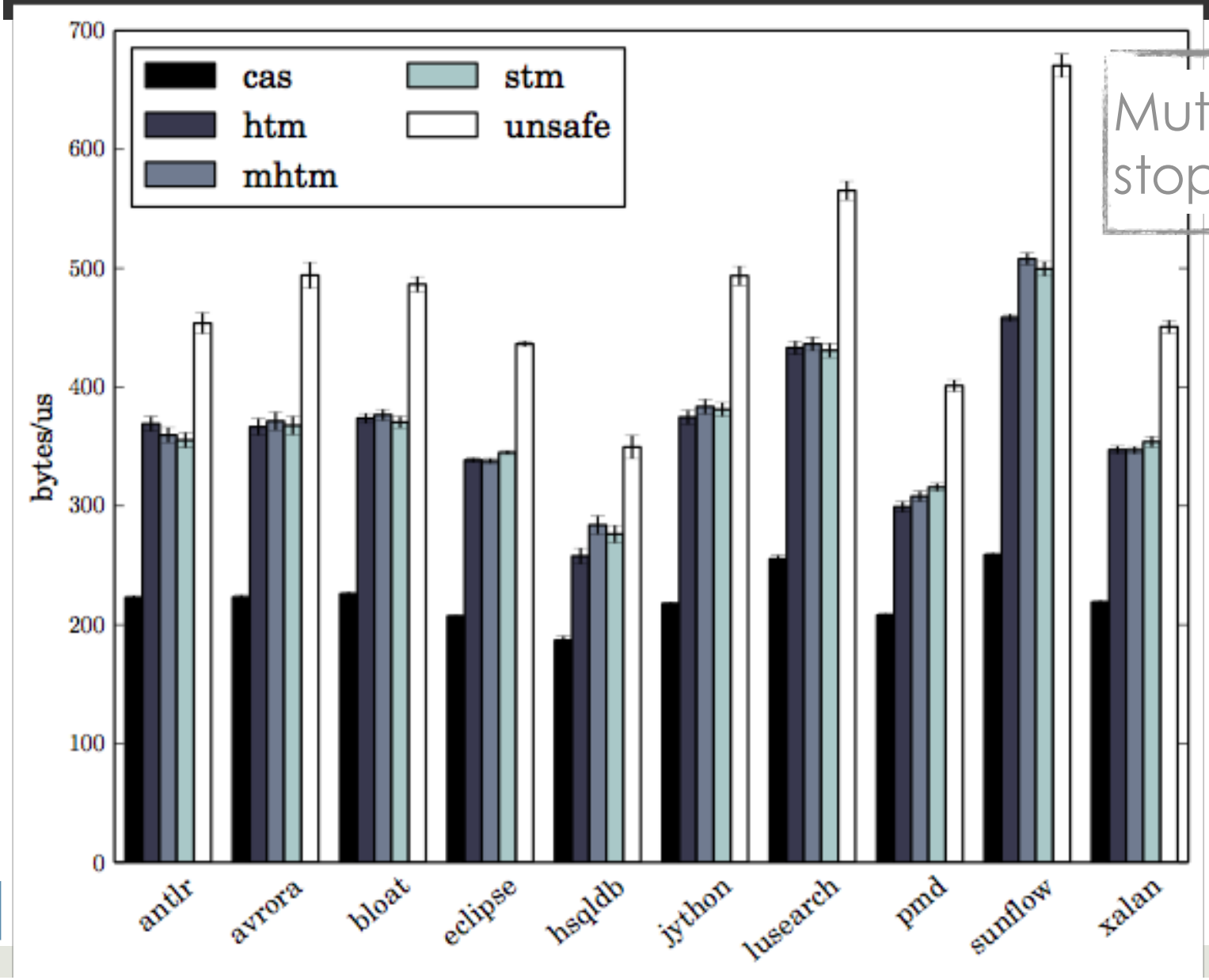
...

MFENCE

verify to-space with from-space

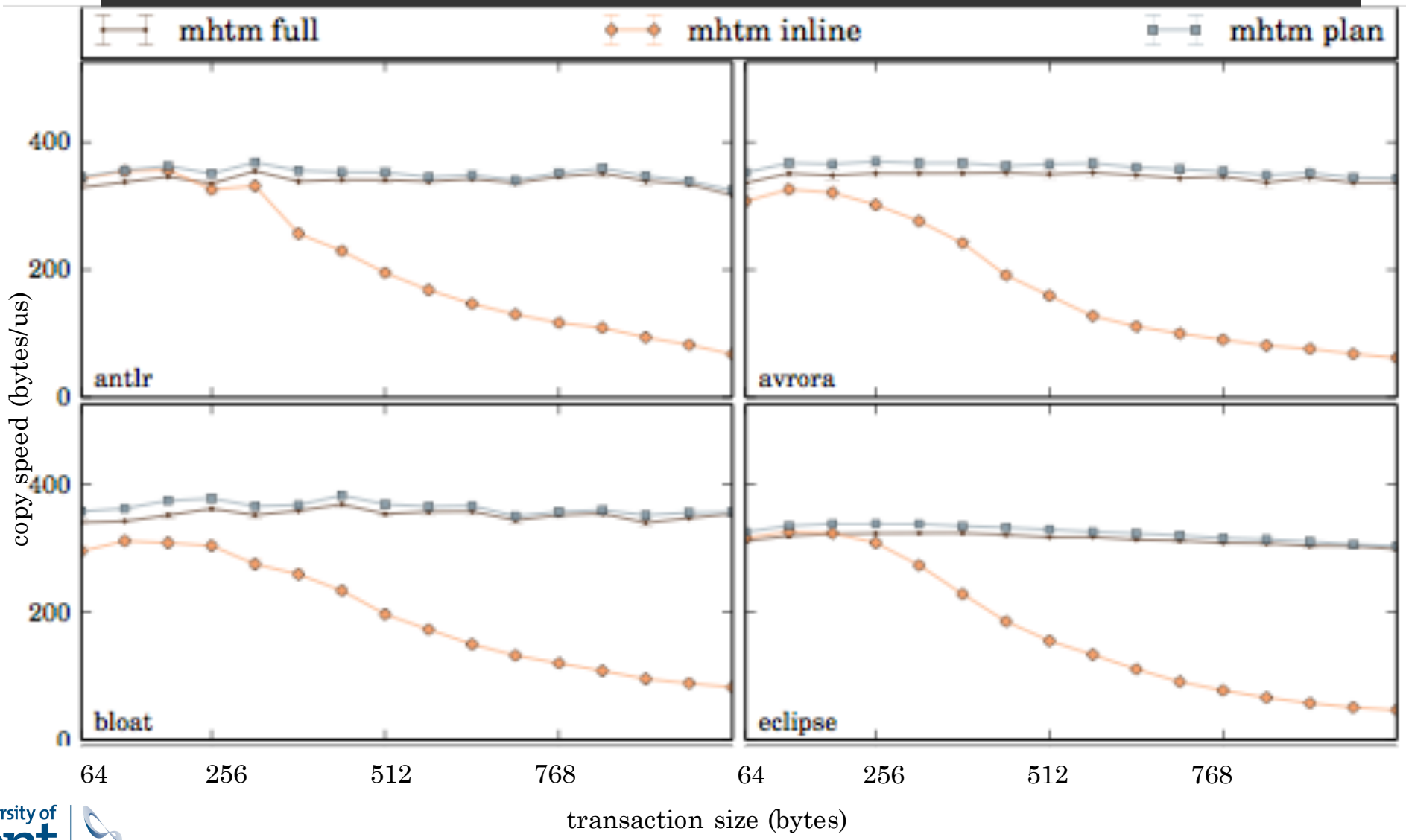
NB. Mutator not reading to-space

Raw copying speed

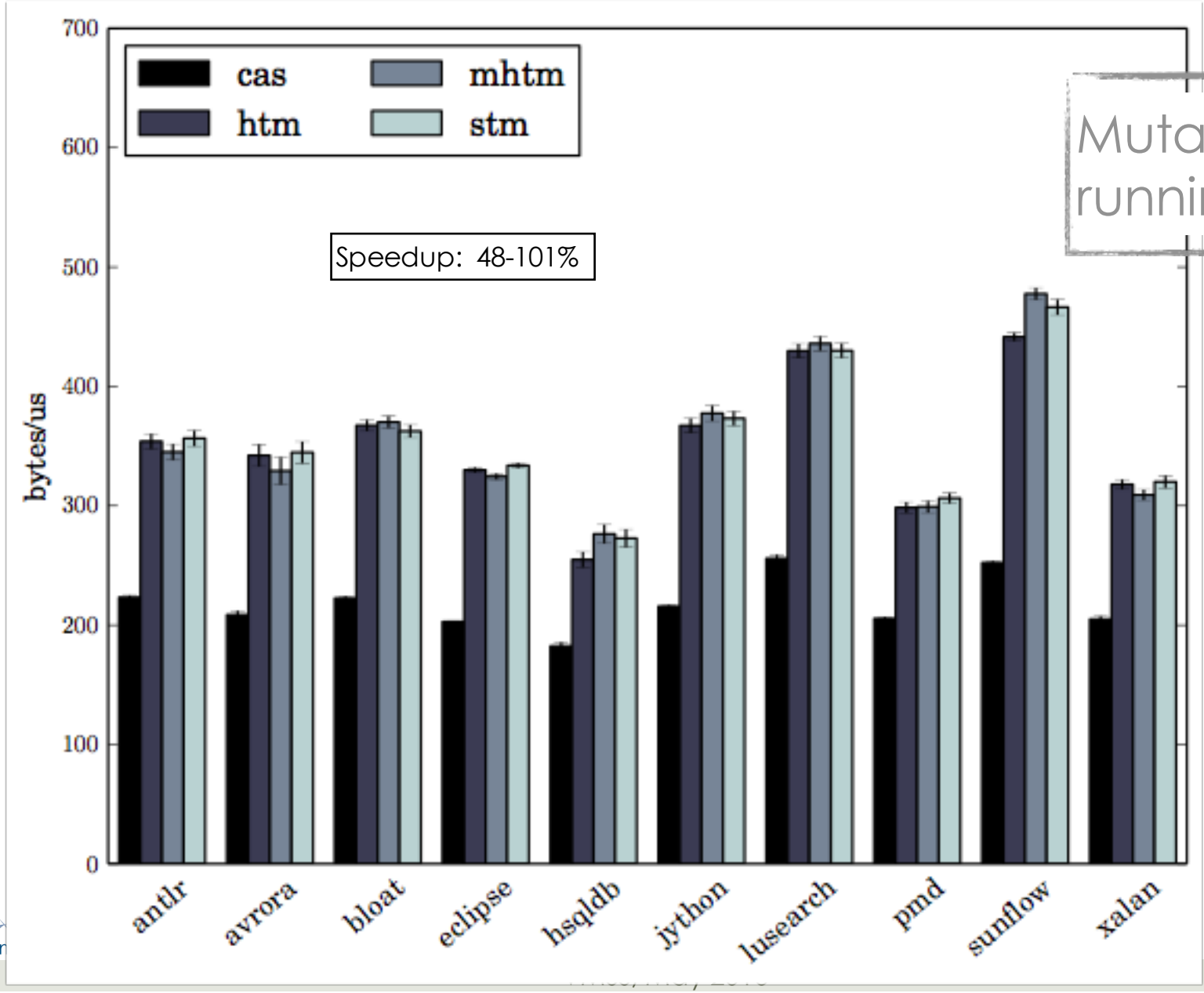


Mutators stopped

Copy speed v transaction size



Concurrent copying speed



BUT IS IT CORRECT?

Assertions + Sanity checking

- Assertions
  - Assertions are the GC developer's friend
  - Use copiously
- Sanity checking
  - Jikes RVM provides a sanity checker
  - Use after a GC e.g.
  - Is every reference valid?
  - Or do some point into reclaimed space?

Testing

- Assertions and sanity checking are just a form of testing.
- Non-determinism.
  Concurrently scheduled threads.
  Collections at different times.
  Relaxed memory.
- How many invocations to give confidence of correctness?
  10? 100? 500? 1,000?
- What's the measure of "correct"?
  Unmodified Jikes RVM will not succeed 1000/1000 times.
  My version fails no more often than the unmodified version??

MODEL CHECKING

Bounded Model Checking

- Verification technique for asynchronous process systems
- Verify some property such as an invariant
- Model checker will…
  - visit all possible states reachable from the initial state
  - check whether a given property holds in every state
- ⇒Model must be small — time and space

SPIN

- Specification in the Promela language
- Sequential processes
- Asynchronous communication via channels
- Shared variables
- Properties = Linear Temporal Logic assertions injected into the model

[Gerard Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004]

Model

- Collector thread process
- Mutator thread process
- X86 relaxed memory process
  - CPU stores inserted into its local FIFO store buffer
  - Store-load forwarding: core can see any store sin its store buffer
  - Only the core that issued *a = v can read the latest value v
- Assume cache coherency

CPU model

- Channels between a thread process and the memory process

```
#define MUTATOR_WRITE (a, v)
  atomic {
     mutator_queue ! a,v;
     mutator_local_memory[a] = v;
     mutator_queue_count[a]++
  }

#define MUTATOR_READ(a, v)
  atomic {
     if
       :: mutator_queue_count[a] == 0
          -> v = shared[a]
       :: else
          -> v = mutator_local_memory[a]
     fi
  }
```

Send <a,v> to mutator_queue channel

Store-load forwarding

```
#define COMMIT_WRITE (q, count)
  (len(q) > 0) -> q ? a,v
      -> shared[a] = v; count[a]--
```

Commit write to shared memory

```
active proctype memory() {
  do
    :: atomic{COMMIT_WRITE(mutator_queue,  mutator_queue_count)}
    :: atomic{COMMIT_WRITE(collector_queue, collector_queue_count)}
  od
}
```

Infinitely repeated, non-deterministic choice: which channel to drain

Scenario

- Single object with a single non-reference field.
- Each semi-space contains one object.

Mutator

```
proctype mutator() {
  byte x = 0, y;
  do
    :: true ->
         x = 1 - x;
         MUTATOR_WRITE(fromSpace, x);
         MUTATOR_WRITE(toSpace, x);
    :: true ->
         if
           :: !flipped -> MUTATOR_READ(fromSpace, y)
           :: else -> MUTATOR_READ(toSpace, y)
         fi;
         assert(x == y);
  od;
}
```

Mutator may change a field

Read from the appropriate semispace

Check that the mutator reads what it wrote

Collector: CAS

```
proctype collector() {
  byte r1, r2, tmp;
  do
   ::true ->
     COLLECTOR_READ(fromSpace, r1);
     COLLECTOR_READ(toSpace, r2);
     if
      ::(r1 == r2) -> break
      ::else -> atomic {                          /* CAS */
          COLLECTOR_READ(toSpace, tmp);
          if
            ::(r2 == tmp) -> COLLECTOR_WRITE(toSpace, r1)
            ::else -> break
          fi;
          COLLECTOR_MFENCE
        }
     fi
  od
  COLLECTOR_MFENCE;
  flipped = true;
}
```

Flush the store buffer

Handshake and change phase

Collector: STM

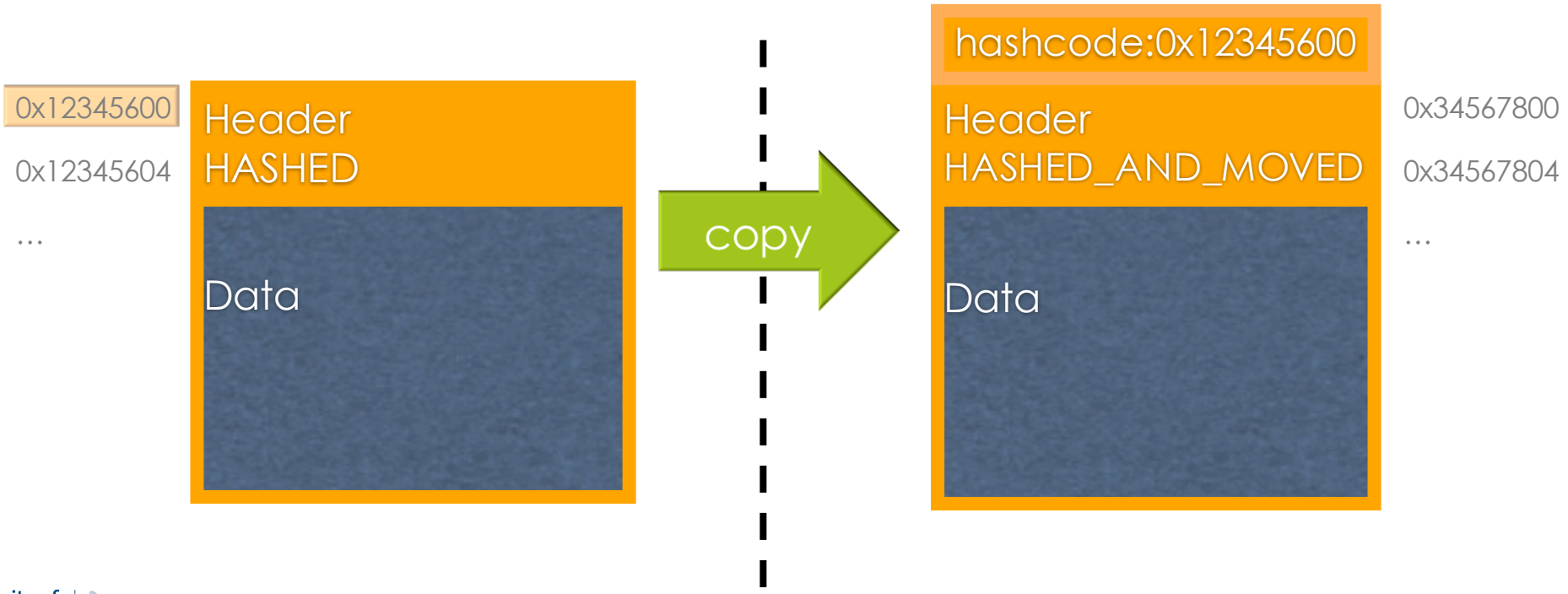
```
proctype collector() {
  byte x, y;
  do
    ::true ->
       COLLECTOR_READ(fromSpace, x);
       COLLECTOR_WRITE(toSpace, x);
       COLLECTOR_MFENCE;
       COLLECTOR_READ(fromSpace, y);
       COLLECTOR_READ(toSpace, x);
       if
         ::(x == y) -> break
         ::else -> skip
       fi
  od;
  COLLECTOR_MFENCE;
  flipped = true;
}
```

verify

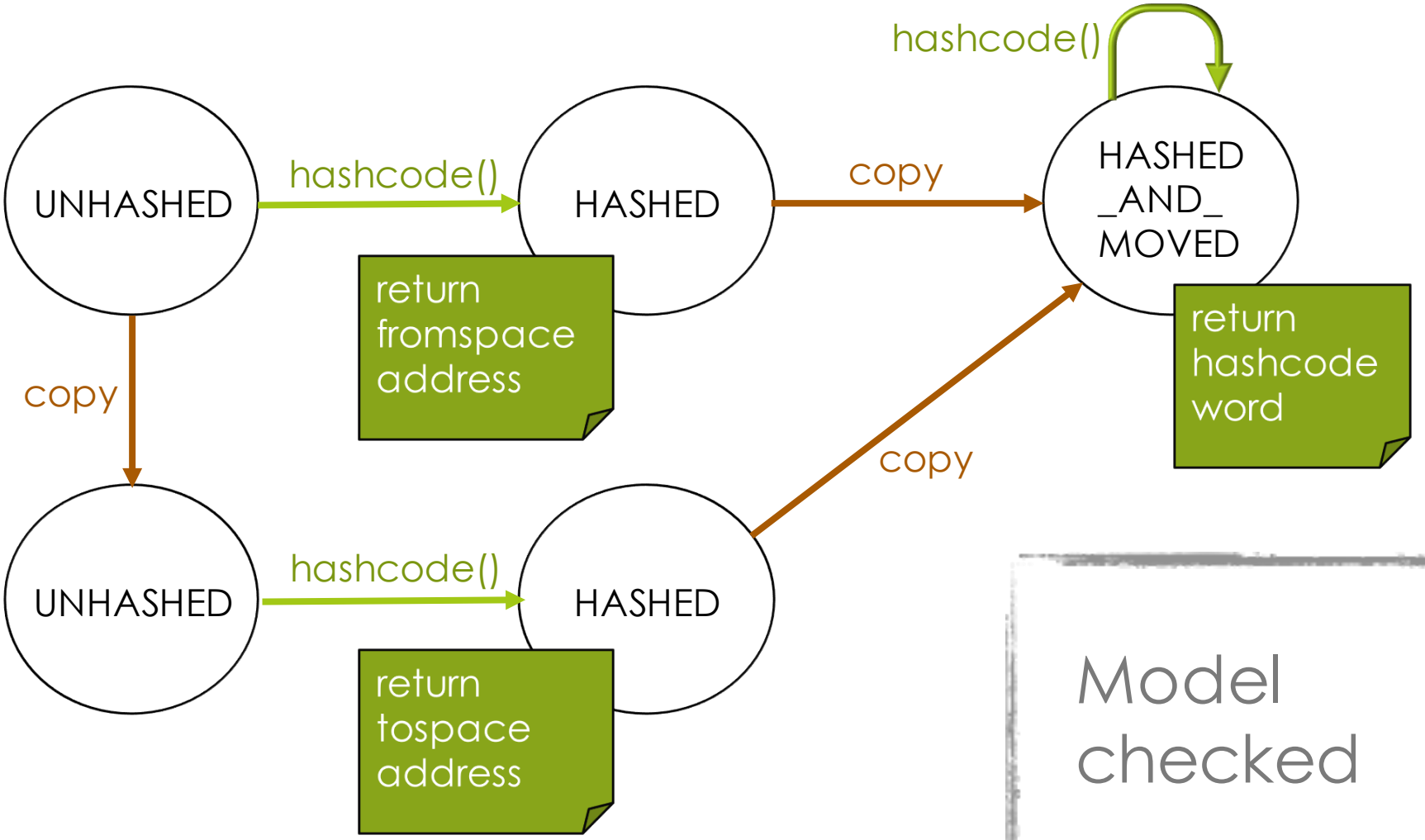
Handshake and change phase

Hashcode()

- hashCode() must consistently return the same integer…
- Address-based hashing: obj.hashcode() = address of obj
- Copying GC moves objects!



Hashing

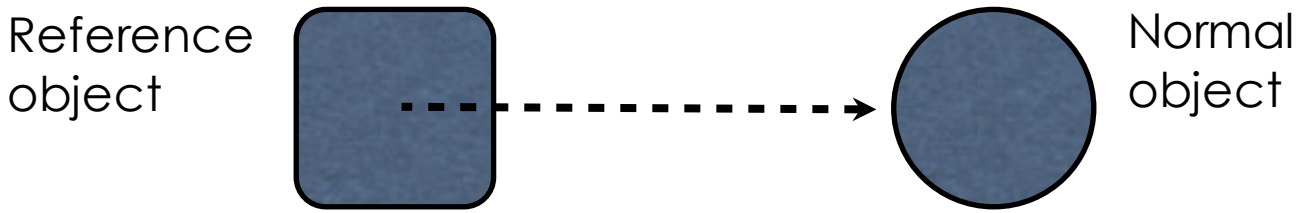


Model checked

Java Reference types

- Strong - usual references.
- Soft - used for caches that the GC can reclaim.
- Weak - used for canonicalised mappings that do not prevent the GC from reclaiming their keys or values.
- Phantom - used for scheduling pre-mortem cleanup actions more flexibly than finalisers.

Reference.get()



- SoftReference.get() returns a strong reference to its target or null if the GC has cleared the referent.
- WeakReference.get() returns a strong reference to its target or null if the GC has cleared the referent.
- PhantomReference.get() always returns null.
- Effectively, get() may make an object that was safe to reclaim unsafe to reclaim!

Reachable objects

- Reachability is defined informally in the java.lang.ref package.
  - Strongly: can be reached without traversing any reference objects.
  - Softly: not strongly reachable but can be reached by traversing a soft reference.
  - Weakly: neither strongly nor softly reachable but can be reached by traversing a weak reference.
  - Phantom: neither strongly, softly nor weakly reachable, it has been finalised, and some phantom reference refers to it.

Reachable objects

- Reachabil… …rmally in the java.lang…
  - Strongly: can be reached without traversing any reference objects.
  - Softly: not strongly reachable but can be reached by traversing a soft reference.
  - Weakly: neither strongly nor softly reachable but can be reached by traversing a weak reference.
  - Phantom: neither strongly, softly nor weakly reachable, it has been finalised, and some phantom reference refers to it.

GC *will not* reclaim

Reachable objects

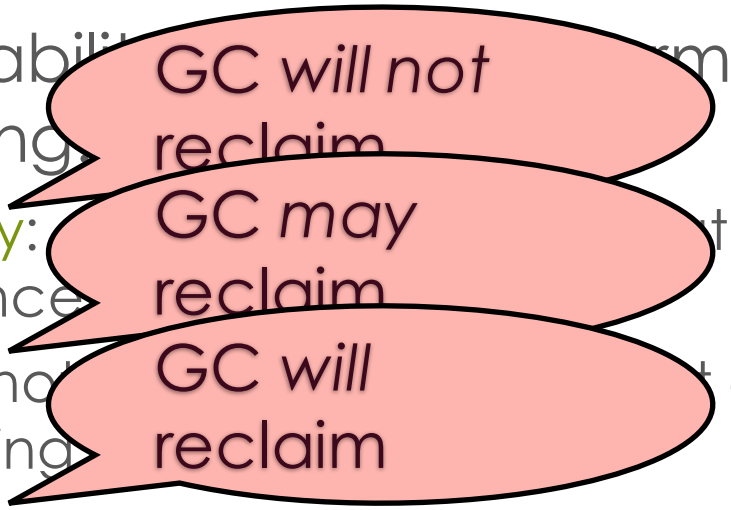
- Reachabil… …rmally in the java.lang…
  - Strongly: … …t traversing any reference…
  - Softly: not strongly reachable but can be reached by traversing a soft reference.
  - Weakly: neither strongly nor softly reachable but can be reached by traversing a weak reference.
  - Phantom: neither strongly, softly nor weakly reachable, it has been finalised, and some phantom reference refers to it.

GC *will not* reclaim

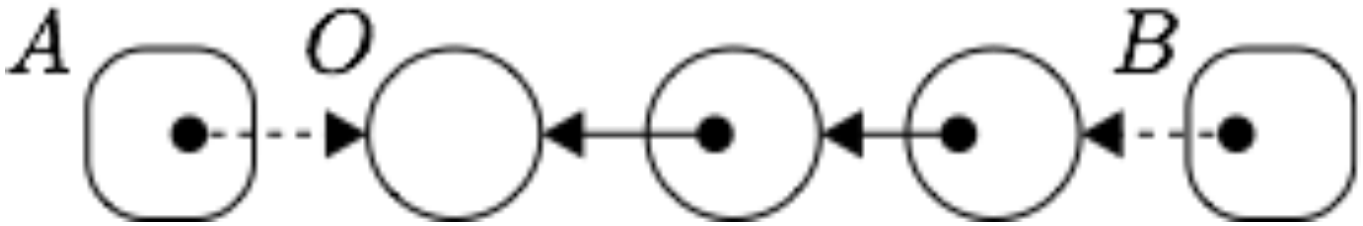
GC *may* reclaim

Reachable objects

- Reachabil… …rmally in the java.lang…
  - Strongly: …t traversing any reference…
  - Softly: no… …t can be reached by traversing…
  - Weakly: neither strongly nor softly reachable but can be reached by traversing a weak reference.
  - Phantom: neither strongly, softly nor weakly reachable, it has been finalised, and some phantom reference refers to it.



Atomicity requirement



- java.lang.ref.Reference.get() returns a *strong* reference
  ⇒ race between collector and mutator.
- If GC decides to reclaim a softly reachable target *O* it must clear *atomically*
  - *all* soft references to *O* (e.g. reference from *A*), and
  - *all* soft references to other softly-reachable objects from which *O* is reachable through a chain of strong references (e.g. reference from *B*).

"Specification"

- "An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference."
- The java.lang.ref definitions are vague, e.g. they do not
  - specify how many soft references we can traverse and when.
  - require that there be no weak or phantom references in the chain.
- Ditto for other reference types...
- There are errors in implementation(s).

Formalisation

- Relations, $R \subseteq \mathbb{P}((\text{Objects} \cup \text{Roots}) \times \text{Objects})$
- e.g. *StrR*, *SoftR*, *WeakR*, *PhantR*

- Transitive closure,
  $\text{TC}(X, R) = \{ o \in \text{Objects} \mid \exists x \in X \,.\, x \, R^* \, o \}$
- e.g. $\mathit{StrongReachable} = \text{TC}(\text{Roots}, \mathit{StrR})$

Correct specification

- $SoftReachable = \mathsf{TC}(\text{Roots}, StrR \cup SoftR) - StrongReachable$

- $WeakReachable = \mathsf{TC}(\text{Roots}, StrR \cup SoftR \cup WeakR) - StrongReachable - SoftReachable$

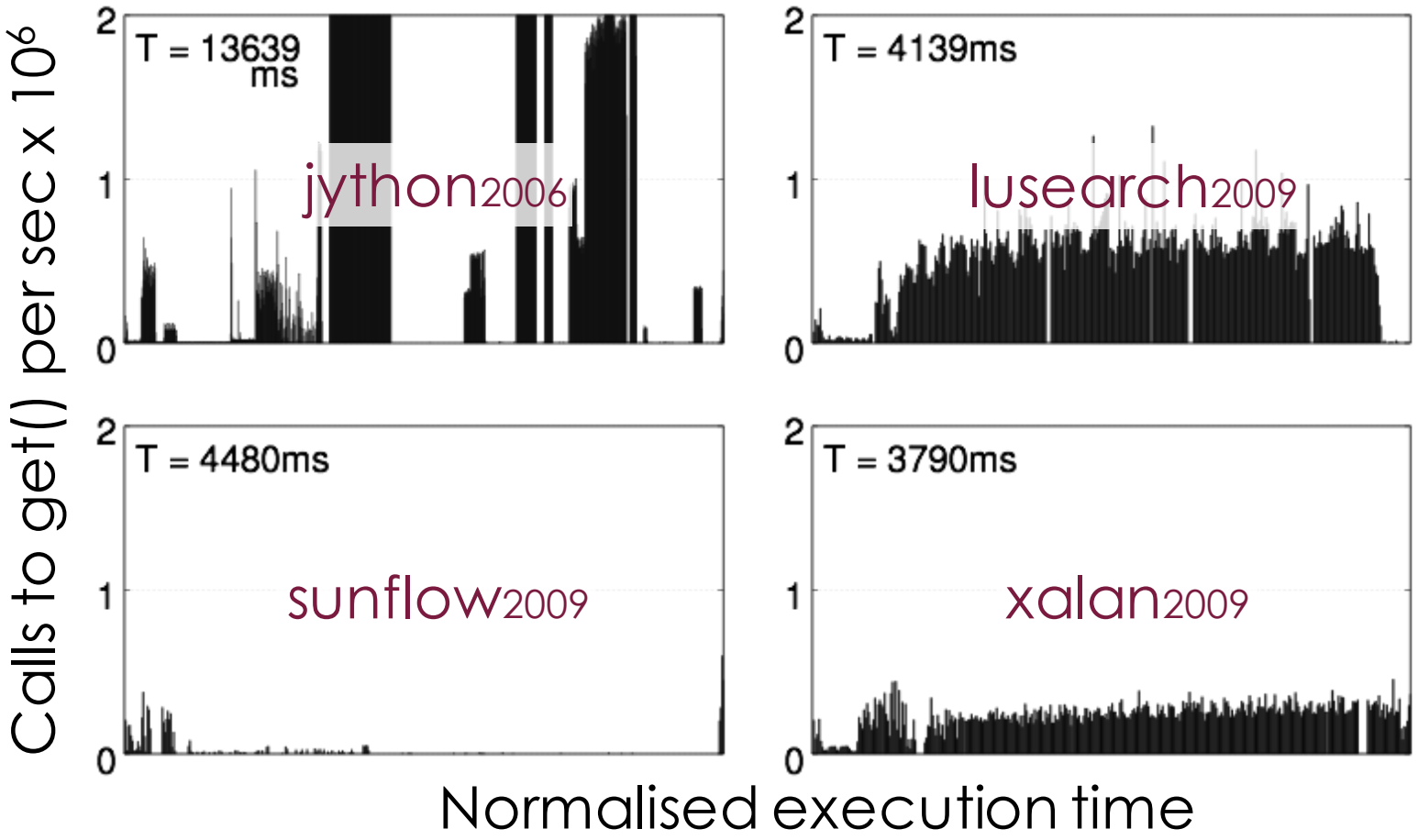
- $PhantomReachable = (\mathsf{TC}(\text{Roots}, StrR \cup SoftR \cup WeakR \cup PhantR) \cap Finalised) - StrongReachable - SoftReachable - WeakReachable$

Clearing references

- If the GC decides to clear a soft (weak) reference to a softly (weakly) reachable object o, it must also clear all soft (weak) references to the sets
- $softToClear(o) = \{\, w \in SoftReachable \mid w\ StrR^*\ o \,\}$
- $weakToClear(o) = \{\, w \in WeakReachable \mid w\ (StrR \cup SoftR)^*\ o \,\}$
- Note

$$phantomToClear(o) = \{\}$$

Reference type usage

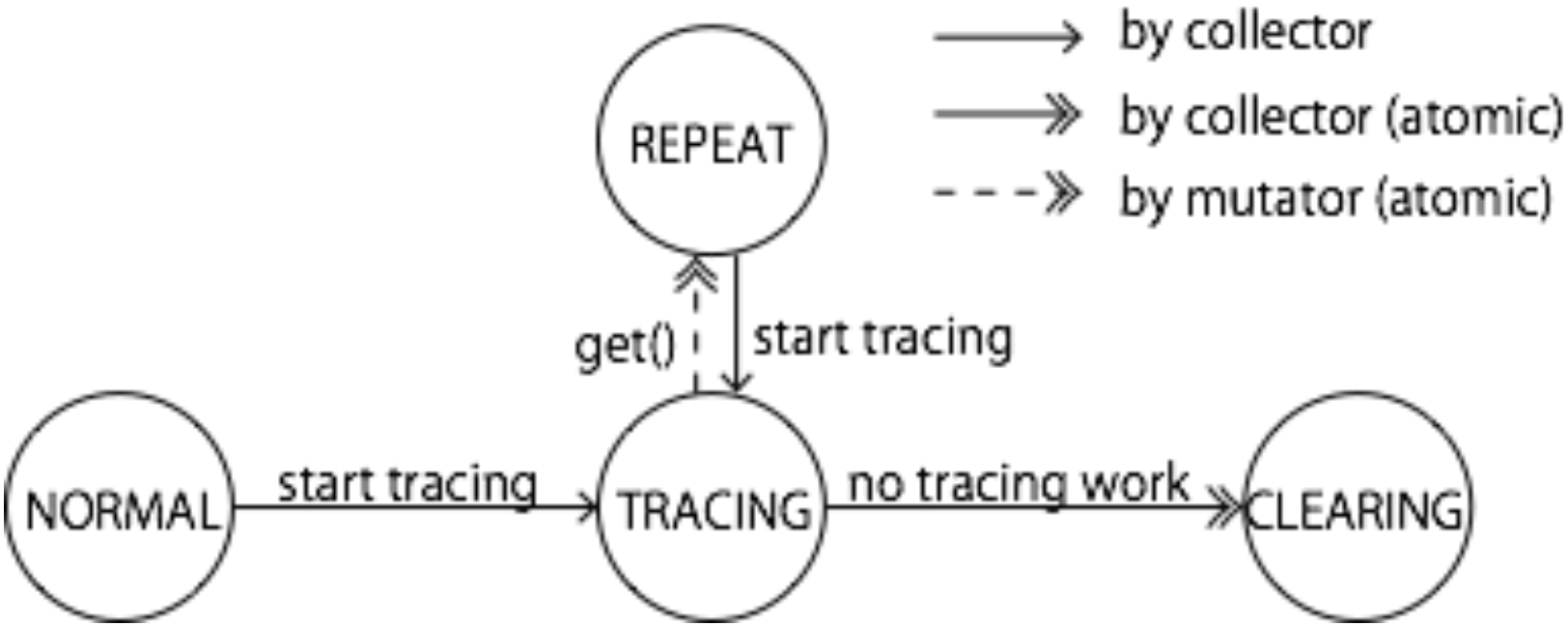


Options for implementation

- To process reference types, we could
  - Stop the world, or
  - Block any mutator that calls get() [Domani et al, 2000] or
  - Process objects on-the-fly, never blocking a mutator other than to scan its roots

[Ugawa et al, ISMM14]

GC State


by collector
by collector (atomic)
by mutator (atomic)
REPEAT
get()
start tracing
NORMAL
start tracing
TRACING
no tracing work
CLEARING

Insertion-barrier code

```
collection() {
  insertionBarrier ← ON;
  transitiveClosureFromRoot();
  while (true) { /* try to terminate */
    refState ← TRACING;
    handshake();
    transitiveClosureNoRootScan();
    scanRoot();
    if (workQueue.empty() &&
       CAS(refState, TRACING, CLEANING))
    break;
  }
  insertionBarrier ← OFF;
  clearReference();
  refState ← NORMAL;
  handshake();
  reclaim();
}
```

```
get() {
 while (true) {
   switch(refState) {
     case NORMAL: case REPEAT:
        return referent;
     case TRACING:
        if(referent=null  | |  COLOR(referent)≠WHITE)
           return referent;
        CAS(refState, TRACING, REPEAT);
        break; /* retry */
      case CLEANING:
        if(referent=null  | |  COLOR(referent)≠WHITE)
          return referent;
        return null;
    }
  }
}
```

Insertion-barrier code

```
collection() {
  insertionBarrier ← ON;
  transitiveClosureFromRoot();
  while (true) { /* try to terminate */
    refState ← TRACING;
    handshake();
    transitiveClosureNoRootScan();
    scanRoot();
    if (workQueue.empty() &&
      CAS(refState, TRACING, CLEANING))
    break;
  }
  insertionBarrier ← OFF;
  clearReference();
  refState ← NORMAL;
  handshake();
  reclaim();
}
```

```
get() {
 while (true) {
  switch(refState) {
    case NORMAL: case REPEAT:
      return referent;
    case TRACING:
      if(referent=null || COLOR(referent)≠WHITE)
        return referent;
      CAS(refState, TRACING, REPEAT);
      break; /* retry */
   case CLEANING:
      if(referent=null || COLOR(referent)≠WHITE)
```

No guarantee of termination

```
collection() {
  insertionBarrier ← ON;
  transitiveClosureFromRoot();
  deletionBarrier ← ON;
  handshake();
  scanRoot();
  insertionBarrier ← OFF;
  while(true) {
    refState ← TRACING;
    handshake();
    transitiveClosureNoRootScan();
    if(workQueue.empty() &&
      CAS(refState, TRACING,
CLEANING))
      break;
  }
  deletionBarrier ← OFF;
  clearReferences();
  refState ← NORMAL;
  handshake();
  reclaim();
}
```

```
get() {
  while(true) {
    switch(refState ) {
      case NORMAL:
        return referent;
      case REPEAT:
        if (referent=null || COLOR(referent)≠WHITE)
          return referent;
        COLOR(referent) ← GREY;
        return referent;
      case TRACING:
        if (referent=null || COLOR(referent)≠WHITE)
          return referent;
        if (CAS(refState, TRACING, REPEAT)) {
          COLOR(referent) ← GREY;
          return referent;
        }
        break; /* retry */
      case CLEANING:
        if (referent=null || COLOR(referent)≠WHITE)
          return referent;
        return null;
}}}
```

Deletion-barrier code

```
collection() {
  insertionBarrier ← ON;
  transitiveClosureFromRoot();
  deletionBarrier ← ON;
  handshake();
  scanRoot();
  insertionBarrier ← OFF;
  while(true) {
    refState ← TRACING;
    handshake();
    transitiveClosureNoRootScan();
    if(workQueue.empty() &&
      CAS(refState, TRACING,
CLEANING))
      break;
  }
  deletionBarrier ← OFF;
  clearReferences();
  refState ← NORMAL;
  handshake();
  reclaim();
}
```

```
get() {
  while(true) {
    switch(refState ) {
      case NORMAL:
        return referent;
      case REPEAT:
        if (referent=null || COLOR(referent)≠WHITE)
          return referent;
        COLOR(referent) ← GREY;
        return referent;
      case TRACING:
        if (referent=null || COLOR(referent)≠WHITE)
          return referent;
        if (CAS(refState, TRACING, REPEAT)) {
          COLOR(referent) ← GREY;
          return referent;
```

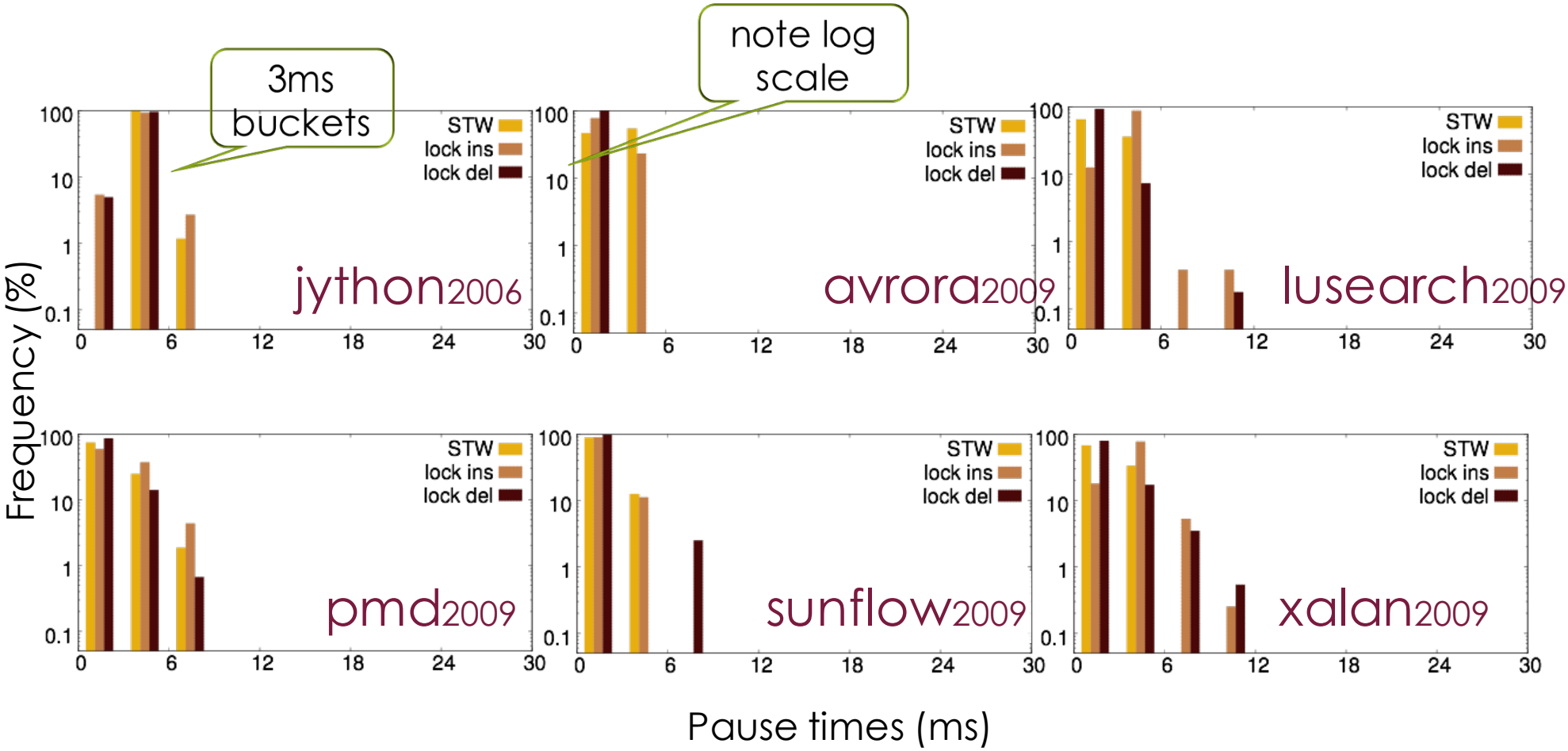
R(referent)≠WHITE)

Termination guaranteed

Deletion-barrier code

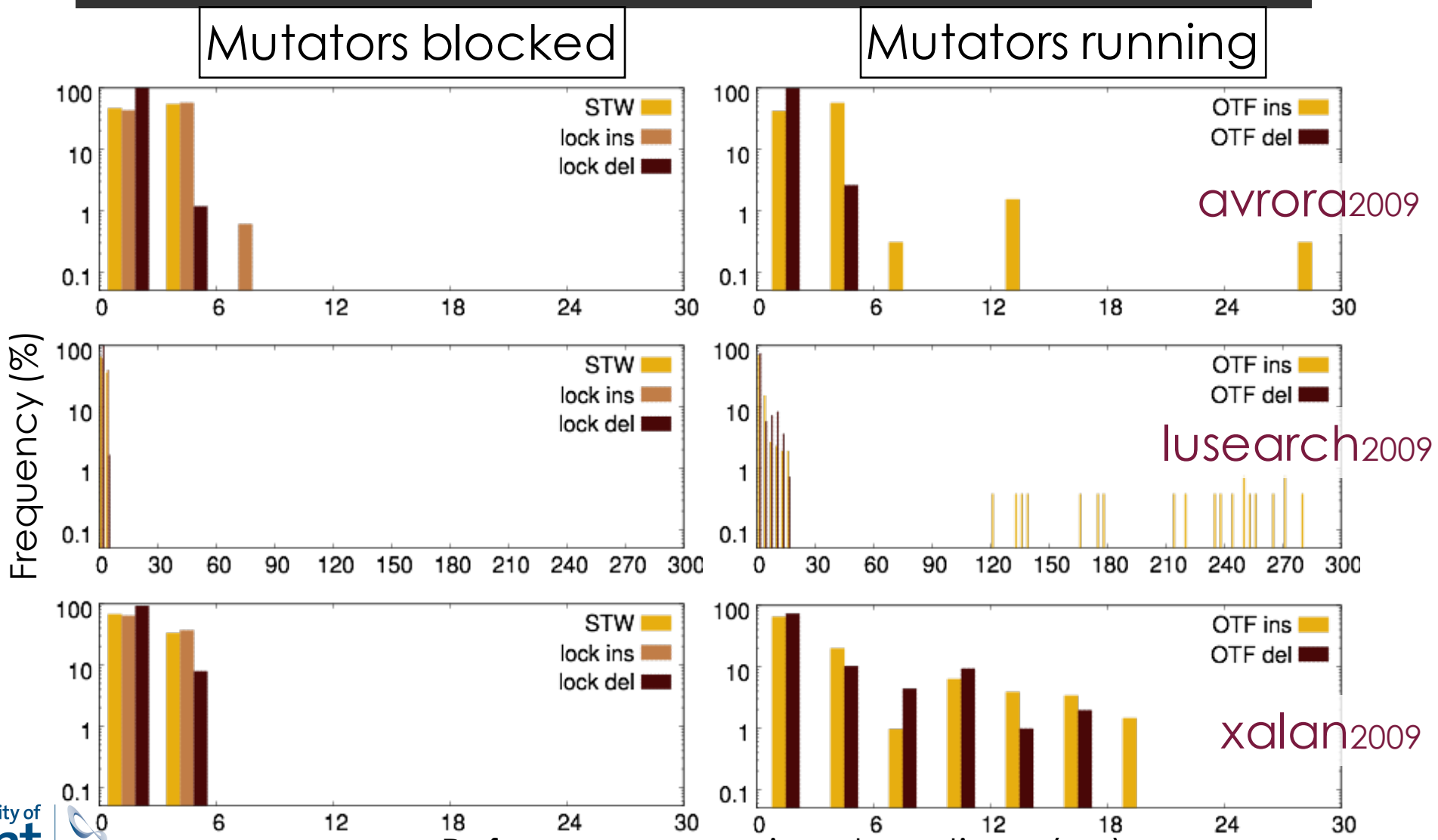
REFERENCE TYPE PROCESSING EVALUATION

Pause time distribution blocking methods

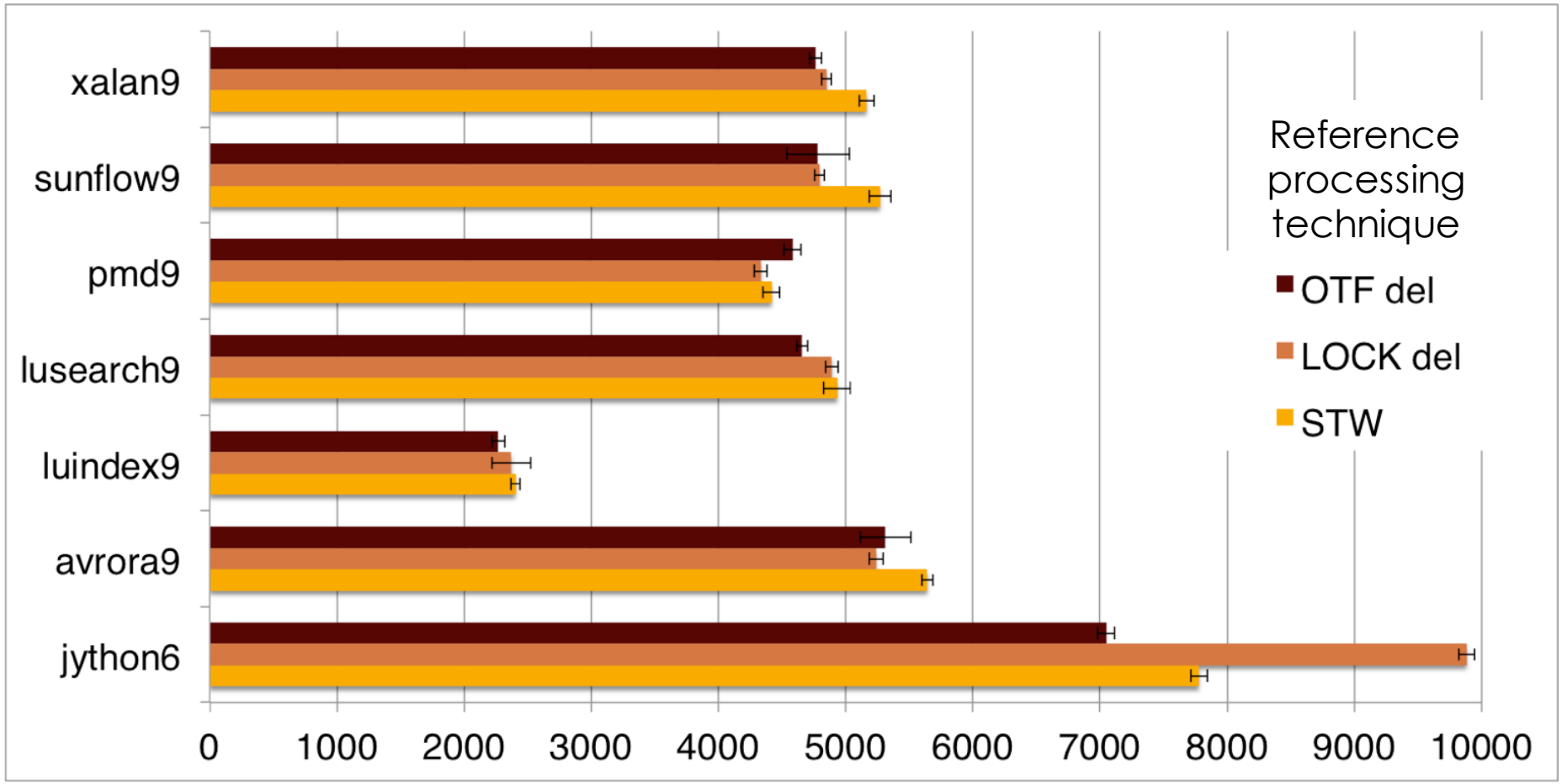


Jikes RVM Sapphire GC; 4-core, 3.4GHz Core i7-4770; Ubuntu Linux 12.04.4

Reference processing phase times



Execution times



Termination without STW

- While GC is TRACING, mutator.get() may make a weakly reachable white object strongly reachable
- Insertion barrier
  - repeatedly scan roots and trace until work queue is empty
  - If TRACING, get() white referent sets state atomically to REPEAT and retries get()
- Deletion barrier
  - If TRACING, get() white referent sets state atomically to REPEAT and retries
  - Shades white referent grey in both REPEAT and TRACING states

Model checking termination

- Properties
  - P1: Safety A mutator will never see a reclaimed object.
  - P2: Consistency Once any get() method called on a reference object returns null, a mutator will never see the referent of that object.
  - P3: Termination GC eventually terminates.
- Model checking
  - P1, P2 hold
  - P3 does not hold for the insertion barrier solution
  - P3 does hold for the deletion barrier solution

Model checking: evaluation

- Practical.
- Easy to construct and check models.
- Bounded model checking cannot give a 100% guarantee of correctness…
- …but gives confidence.
- Concurrent garbage collection is complex and it is easy to overlook corner cases.
- Model checking offers reasonable confidence in an algorithm at a reasonable cost.

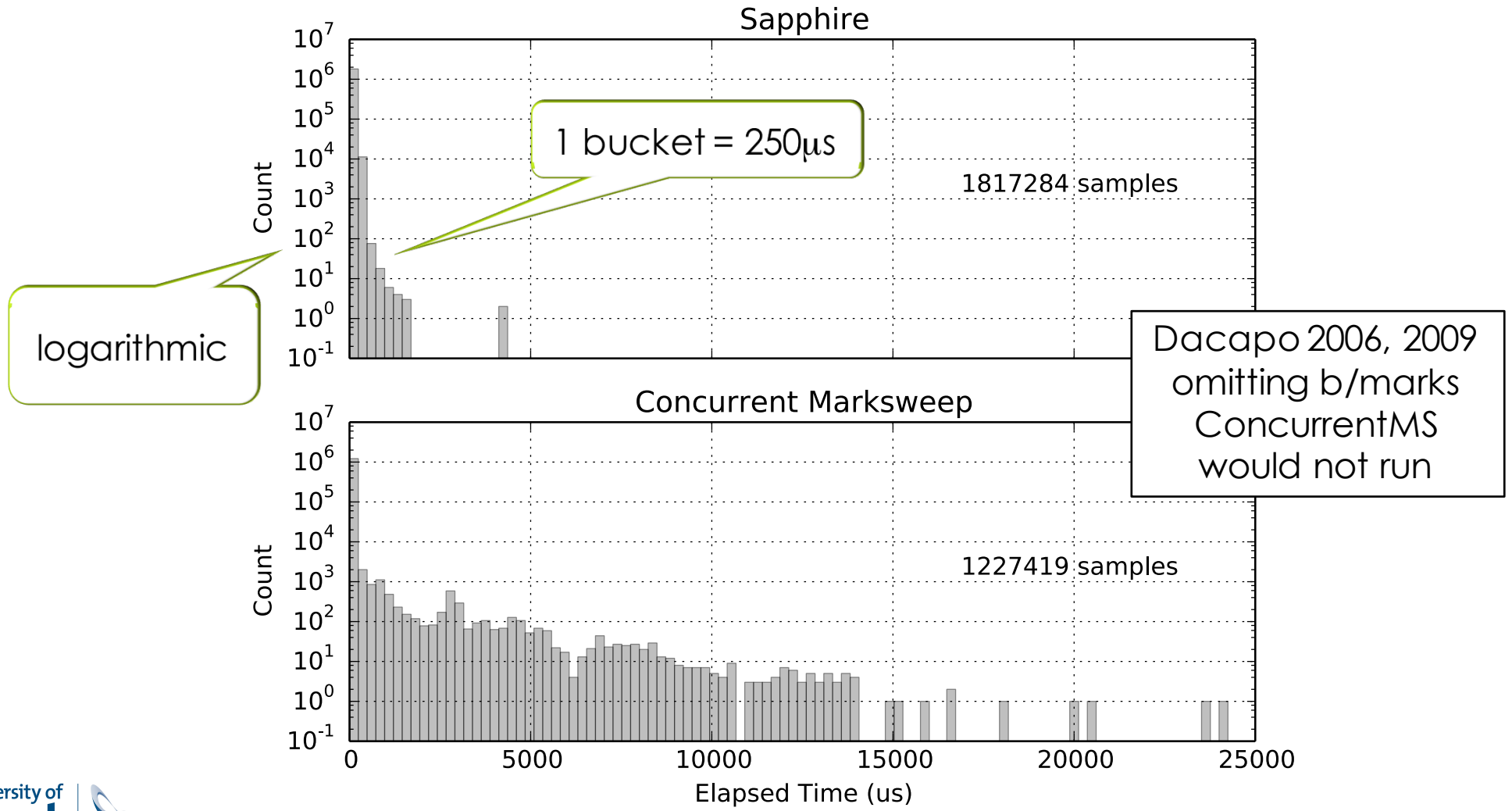
Punchline

- Formal specification of reference type behaviour.
- Model-checked implementation.
- OTF reference processing phases are longer in the worst case but, with deletion barriers, not by much.
- Overall execution time is not increased significantly by processing references OTF, and is often reduced.

Summary

- On-the-fly copying collection for a full JVM is extremely complex.
- Abstractions and invariants are the key to comprehension
- Model checking is a practical way to provide confidence in algorithms

Distribution of task times



Dacapo 2006, 2009 omitting b/marks ConcurrentMS would not run

Verification

- *Automated Verification of Practical Garbage Collectors*, Chris Hawblitzel and Erez Petrank. POPL09.
- *A Certified Framework for Compiling and Executing Garbage-Collected Languages*, Andrew McCreigh, Tim Chevalier and Andrew Tolmach. ICFP10.
- *Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO*, Peter Gammie, Tony Hosking and Kai Engelhardt. PLDI15.